fsm2 – A Scripting Language for Manipulating Weighted Finite-State Automata

User Manual

Thomas Hanneforth

version 1.0.0 beta
12.01.2011

fsm2 – A Scripting Language for Manipulating Weighted Finite-State Automata

Table of Contents

INTRODUCTION 3
COMMAND LINE ARGUMENTS 3
START-UP FILES 3
COMMAND/ FILE NAME COMPLETION 3
FSM2 COMMANDS 4
GENERAL 4
STACK COMMANDS 5
SETTINGS RELATED COMMANDS 6
INPUT/OUTPUT 8
COMMANDS RELATED TO SYMBOL SPECIFICATIONS 12
VERSION RELATED COMMANDS 12
FSM ALGEBRA 13
UNARY COMMANDS 13
BINARY COMMANDS 14
OPTIMIZATION AND CONVERSION 16
COMMAND RELATED TO STATISTICAL MODELLING 18
COMMANDS RELATED TO PATTERN MATCHING 19
COMMANDS RELATED TO DISTANCE COMPUTATIONS 20
TEST COMMANDS 21
SEMIRING COMMANDS 22
SEMIRINGS 23
A NOTE ON THE STRING AND UNIFICATION SEMIRINGS 24
SEMIRING TYPES 25
THE UNIFICATION (FEATURE STRUCTURE) SEMIRING 25
USING MACROS 27
COMMANDS RELATED TO MACROS 27
SOME NOTES ON MACROS 28
USING VARIABLES 31
COMMANDS RELATED TO VARIABLES, ENCODINGS AND SUBSTITUTIONS 31
LANGUAGE MODELING 32
CLASS-BASED LANGUAGE MODELS 33
FSM<2.0> REGULAR EXPRESSION SYNTAX 35
REGULAR EXPRESSION PRECEDENCE TABLE 39

FILE FORMATS 40
SYMBOL SPECIFICATIONS 41
BASIC FORMAT OF A SYMBOL SPECIFICATION 41
SPECIAL TYPES AND SYMBOLS 42
SPECIAL SUPERTYPES 44
SPECIAL TERMINAL SYMBOLS 44
OTHER FEATURES 46
GENERAL LEXICON FILES 47
"ACYCLIC" LEXICON FILES 48
GRAMMAR RULES FILES 50
GRAMMAR APPROXIMATION 51
DICTIONARY REWRITING FILES 52
TEXTUAL FSM FILES 53
EXAMPLES 54
LIMITATION AND KNOWN BUGS 62
EXIT CODES 62

Introduction

fsm2 is a simple XFST-style interpreter for FSM<2.0>. Like XFST[1] it is based on a stack machine, that is, most *fsm2*-commands manipulate finite state machines on a stack.

There are yet some differences to XFST:

1. All higher-level operations in FSM<2.0> (on regular expressions, grammars, lexicons etc.) are based on a symbol specification. So, the first step in *fsm2* in almost every case will consist in loading a symbol specification file.
2. There are some additional commands related to language modelling, distance computations and longest-match rewriting.
3. The non-commutative operations concatenation, composition, difference and cross product will work in the opposite direction compared with XFST: if two FSMs A and B are on the stack with A being topmost, composition will compute B x A. At the moment, this behaviour seems advantageous to me, but I may be wrong and will change this.
4. All binary operations will manipulate only the two topmost FSMs on the stack, not all as in XFST.

Command line arguments

Currently, *fsm2* can be executed in interactive or script mode:

`$ fsm2`	Starts *fsm2* in interactive mode
`$ fsm2 SCRIPTFILE`	Starts *fsm2* in script mode. The script in SCRIPTFILE is processed and the interpreter is exited afterwards. The default file extension for script files is .fsm2.

Start-up files

If the home directory[2] contains a file `.fsm2.ini`, the commands in that file will be executed right after start-up.

If additionally the current working directory contains also a file `.fsm2.ini`, it will also be executed after start-up.

Command/ file name completion

On Linux systems, *fsm2* supports a (context-dependent) completion mechanism (based on the GNU *readline* library). By typing a prefix of a command or filename and pressing the TAB key twice, the system will either complete the input (if the prefix is unique) or offer a list of possible continuations.

[1] XFST is a program by XEROX.

[2] Under Linux, the home directory is determined using the $HOME environment variable (normally /home/username). Under the Windows platform, the concatenation of the contents of the %HOMEDRIVE% and %HOMEPATH% variables is used.

fsm2 commands[3]

General

Command	Description
`help [TOPIC]`	Outputs a short command description

`echo ["STRING" …]`
`echo ["STRING" …] > "FILE"`
`echo ["STRING" …] >> "FILE"`
	Write all "STRING" arguments to the console. If no argument is given, output a newline. Note that echo with a string argument does not output a newline. If "FILE" is specified the output strings will be written (>) or appended (>>) to "FILE".

`echoln ["STRING" …]`
`echo ["STRING" …] > "FILE"`
`echo ["STRING" …] >> "FILE"`
	Same as `echo`, but always outputs a newline character at the end.

`system ["STRING"]`	Execute the operation system command given by `STRING`. If `STRING` is omitted, a platform-dependent shell is opened (`bash` on Linux and `cmd` on Windows)
`ls ["STRING"]`	List directory content
`cd STRING`	Change the working directory to STRING
`pwd`	Print the current working directory
`save-history [FILENAME]`	Prints the interactively typed commands in the current *fsm2* session. Note that under Linux/MacOSX a hidden file `.fsm.history` is automatically created in the current directory. This file contains all typed-in commands of all sessions in that directory.

`quit`
`exit`
`bye`
`halt`
	Exit the interpreter

[3] FILE = a valid filename, REGEX = a regular expression, STRING = an ASCII string, SYMBOL = a defined symbol from the symbol specification, SUPERTYPE = a super type from the symbol specification, VAR = a defined variable. Modifiers in [] are optional, | means disjunction.

Stack commands

`print stack`	Print the current content of the FSM stack
`clear stack`	Removes all FSMs from the stack. Clears also the "error occurred" flag.
`clear`	Removes all FSMs from the stack. Clears also the substitution and encoding mappings and the "error occurred" flag. The macro definitions are not cleared. See also: `substitute`, `encode`, `decode`
`pop stack`	Remove the topmost FSM from the stack
`flip stack`	Exchange the two topmost FSMs on the stack
`turn stack`	Reverse the whole FSM stack

Settings related commands

`continue-on-error on|off`	[default: off] If on, script execution continues after a severe error. **Note that the "error occurred" flag which triggers the abortion of the script execution, is reset by the commands `clear stack` and `source`.**
`verbose on|off`	[default: off] If verbose is on, every command is written to the console prior execution
`quiet on|off`	[default: off] If quiet is on, the system doesn't output messages.
`fn-completition on|off`	[default: on] If on, file names will be completed with the default file extension.
`text-format xml|att`	[default: xml] Change the text format used by `print fsm` and `load fsm`. The argument `xml` specifies XML as standard format, while `att` establishes the AT&T format as the standard text format.
`use-symbol-names on|off`	[default: off] Determines whether text input/output with `load fsm` and `print fsm` uses symbol names instead of numbers (only if `text-format=att`)
`use-categories on|off`	[default: on] When on, the commands `print words` and `lookup` will output categories in the format feat=val. When off, feature values will be outputted as normal symbols.
`collect-weight-mode on|off`	[default: off] Affects the output of `print words`. If `collect-weight-mode` is on, weights of equal strings/string pairs are abstractly added. Note that this also affects the number of strings that can be outputted, since collecting weights implies that all strings must be hold in memory.
`approx-delta ["FLOATCONST"]`	[default: 0.0001] Outputs or changes the approximation value *d* for numerical semirings. If the difference of two floating point numbers is less than *d*, the two numbers are treated as being equal. **Note that this option has an impact on the efficiency of operations which rely on distance computations (for example, best-path, minimize, rmepsilon). The approximation value *d* should be set depending on the semiring: smaller in case of the real semiring, bigger for the log semiring.**

`float-precision INT`	[default: 8] The `INT` parameter determines the floating point precision of the `print fsm` command (both AT&T and XML formats). Note that the float precision for dot (`draw` command) output is hardcoded with value 3.
`draw-font "FONTNAME"`	[default: Times New Roman] Specifies the font used for dot output. See `draw` command.
`dot-latex-mode on|off`	[default: off] If `on`, the `draw` command produces dot-files to be processed with the `dot2tex` command (see http://www.fauskes.net/code/dot2tex/) [experimental]
`lookup-unique-analyses on|off`	[default: on] `lookup-unique-analyses` determines whether analyses with the same weights are counted as several analyses. See also: `lookup-max-analyses`
`lookup-max-analyses N`	[default: unlimited (-1)] Sets the max. number of analyses outputted by `lookup` and `lookup-file`. `lookup-unique-analyses` determines whether analyses with the same weights are counted as distinct analyses. See also: `lookup-unique-analyses`
`compress-binary-files on|off`	[default: on] Affects the behaviour of the `save fsm` command. If set to on, binary *fsm2* files will be compressed before they are saved to a file. Note that not every binary *fsm2* format supports compression. See also: `save fsm` , `convert`
`settings`	Shows the current settings.
`sysinfo`	Show size information about states, symbols and weights

Input/Output

`source FILE` `include FILE`	Read in FSM<2.0> script in "FILE". The default file extension for FSM<2.0> scripts is `.fsm2`.
`load fsm FILE [binary] [transducer]`	Load an FSM from a file named `FILE`. If `binary` is omitted this file must be in XML or AT&T format as written by the `print fsm` command. If `binary` is specified FILE must be in one of the binary file formats defined by FSM<2.0> (see `save fsm` command). *Note*: if `FILE` is in AT&T format and represents a transducer, the keyword `transducer` MUST be specified (since the AT&T format is underspecified) If `FILE` is in XML-format, the current semiring (see section Semiring commands) must match the semiring specified in the XML file. Also the internal symbol and state sizes must be compatible with the ones stated in the XML file) See also: `text-format, print fsm, save fsm`
`load regex FILE [verbatim]`	Load the regular expression in `FILE`. Note that only the first line in `FILE` is read in. If `verbatim` is specified, the special regex syntax is deactivated and spaces are treated as regular symbols. See section *File Formats*.
`load grammar FILE [approximate] [dont-replace-preterminals]`	Load a "context free" grammar in FSMCFGCompiler format from `FILE`. If `approximate` is given, the grammar will be compiled with the approximation algorithm by Mohri & Nederhof. If `dont-replace-preterminals` is given, preterminals (that is, nonterminals that do not have other non-terminals in the grammar rule's right-hand side) are not replaced. The default file extension for grammar files is `.grm`. See section *File Formats*.

`load lexicon FILE [verbatim]`	Load a lexicon in FSMLexCompiler format from `FILE`. The resulting FSM contains the disjunction of these regular expressions which may use all available regular expression operators. This file format is useful for representing lexicons and corpora. The default file extension for lexicon files is `.lex`. If `verbatim` is specified, the special regex syntax is deactivated and spaces are treated as regular symbols. See section *File Formats*. *See also*: `load acyclic-lexicon`
`load acyclic-lexicon FILE`	Load a lexicon in FSMLexCompiler format from `FILE` and compiles in into a minimal FSM. As opposed to the `load lexicon` command, the lexicon in FILE must denote an acyclic FSM. Moreover, no special regex operators except concatenation are allowed (but category/feature-value syntax with underspecification is allowed). For acceptors, the compilation uses a memory-efficient and fast incremental minimization algorithm, thus allowing much bigger lexicons to compile. For transducers, first the lexicon is compiled to a trie which is afterwards minimized. The memory requirements are thus bigger as in the unweighted case. The input lexicon need not be sorted. Semirings with multiple outputs are also supported. *Note*: Note that big lexicons with too much underspecification in the type system may cause the compiler to run out of memory, since internally a disjunctive normal form is created prior to creating the minimized machine. Note also that in case of lexicons denoting transducers the result is not in every case minimal (but nearly minimal), since final weights may in some cases be realized earlier on the path. *See also*: `load lexicon`
`load contextrules FILE`	Load a set of context rules in the FSM<2.0> context rule compiler format from `FILE`. See section *File Formats*.

`regex "REGEX" [verbatim]`	Compile the regular expression (given by `REGEX`) to a FSM and push it on the stack. If `verbatim` is specified, the special regex syntax is deactivated and spaces are treated as regular symbols. See section *FSM<2.0> Regular Expression Syntax*
`map SYMBOL ["REGEX"]`	([unary]) Assign the FSM originating from `REGEX` to `SYMBOL` and add the pair to the substitution map (see `substitute` command). This assignment remains valid until a `clear` command is issued. If `REGEX` is not specified, `SYMBOL` will be mapped to a copy of the FSM on top of the stack. See section *FSM<2.0> Regular Expression Syntax*
`apply "REGEX" [bestpath]`	[unary] Apply `REGEX` to the FST at the top of stack and output the resulting strings on the console. If the FSM on top of the stack is weighted and `bestpath` is specified, only the best result strings are shown. The difference between `apply` and `lookup` is that `apply` accepts arbitrary regular expressions whereas `lookup` is restricted to plain (UTF-8) input strings. On the other hand, `lookup` is much faster than apply.
`lookup "STRING" ["STRING" …]`	[unary] Lookup all `STRINGs` in the FSM on top of the stack and output the strings `STRING` is mapped to. Note that the symbols in `STRING` are interpreted literally; there is no special regular expression syntax.
`lookup-nbest N "STRING" …`	[unary] Lookup the n-best `STRINGs` in the FSM on top of the stack and output the strings `STRING` is mapped to. Note that the symbols in `STRING` are interpreted literally; there is no special regular expression syntax.
`lookup file IFILE [> OFILE]` `lookup file IFILE [>> OFILE]`	[unary] Same as `lookup` above, but lookups up all string in the text file named `IFILE` which contains one word per line. If `OFILE` is specified, the results of lookup are written or appended to `OFILE`. In that case, a second file named `OFILE.noanalysis` is created to which all inputs from IFILE are written/appended for which there are no analyses.
`draw FILE`	[unary] Create a visual representation of the FSM on top of stack in Graphviz (dot) format
`print fsm %VAR% [> FILE]`	[unary] Print FSM named by `%VAR%` in XML or AT&T format to console or `FILE`. See `text-format` command.

`print fsm [> FILE]`	[unary] Print FSM on TOS in XML or AT&T format to console or `FILE`. See `text-format` command.
`save fsm FILE`	[unary] Save the FSM on top of stack to a binary file. Note that currently FSMs based on the string semiring (also as a component semiring) cannot be saved in binary format. Instead the XML format has to be used. See also: `print fsm, load fsm`
`input ["PROMPT"]`	Allows the user to type in a (weighted) regular expression. If the regular expression is enclosed in double quotes, these symbols will be stripped off. `PROMPT` is an optional message prompting the user to enter the regular expression.

`print words [> FILE]`
`print words [>> FILE]`
`language`
`strings`
	[unary] Output the (finite) language of the FSM on top of stack. If `FILE` is specified the output strings will be written (>) or appended (>>) to `FILE`.

`fsminfo [properties]`	[unary] Output information about the FSM on top of stack. If `properties` is specified Boolean properties are shown.

Commands related to symbol specifications

`load symspec FILE [utf-8] [precompiled] [att]`	Load the symbol specification (in FSMSymSpec format) in `FILE`. If `FILE` is in UTF-8 format, add the `utf-8` keyword. The `precompiled` modifier causes `load symspec` to assume a precompiled symbol specification. This format is useful in case of symbol files with several hundred thousand of symbols. If `att` is specified the internal mapping form symbols to numbers is compatible with AT&T's lextools. The default file extension for symbol files is `.sym`, the default file extension for precompiled symbol files is `.psym`. For a description of the format of a symbol specification see *File formats*.
`print symspec`	Outputs the current symbol specification in precompiled format.

Version related commands

`set version "Major.Minor.Build"`	Set the version number of all FSMs subsequently written to files in XML and binary format (commands `print fsm` and `save fsm`). All three parts of the version number are integers.
`print version`	Print the version of the FSM on top of stack

FSM Algebra[4]

All unary commands manipulate (replace) the FSM on top of the FSM stack.
All binary commands manipulate (replace) the two topmost FSMs. If the operation is commutative the first operand will be the second FSM on the stack.

Unary commands

`star` `closure`	[unary] Compute the star closure of an FSM
`plus`	[unary] Compute the plus closure of an FSM
`reverse`	[unary] Reverse an FSM, that is reverse its language
`complement [ALPHABET]`	[unary] Compute the complement of an deterministic, unweighted acceptor. If `ALPHABET` – which must be a supertype in the symbol specification – is given, the set of subtypes of `ALPHABET` will be used for complementation. By default, the alphabet used corresponds to the special `<sigma>` supertype.
`complete`	[unary] Add a sink state s with a Σ-loop and add transitions from q to s for every state q with every symbol for which there is no outgoing transition. Operand must be a deterministic, unweighted acceptor. Note: the sink state s will be a final state
`project 1` `project 2`	[unary] Project the input/output tape of a FST
`substitute`	[unary] Applies the currently defined substitutions (see `map` command) to the FSM on top of stack.
`ignore "SYMBOLSTRING"`	[unary] Ignore all symbols in `SYMBOLSTRING`
`bestpath`	[unary] Construct an FSM which represents the best path
`optional`	[unary] Construct an FSM which also accepts ε
`encode {weights, labels}`	[unary] Encodes the FSM on top of the stack. That means that input/output labels and/or weights are mapped to a single new symbol. This mapping is stored in an internal data structure and is used by `decode`.
`decode {weights, labels}`	[unary] Decodes FSM on top of the stack.

[4] FSA = Finite state acceptor, FST = Finite state transducer, FSM = FSA or FST

Binary commands

`concat [N]`	[binary] Compute the concatenation of two FSMs. Note that the operand on top of the stack will be the second operand of the concatenation. If a number *N* (> 1) is specified, the topmost *N* FSMs are concatenated.
`union [N]`	[binary] Compute the union (disjunction) of the two topmost FSMs. If a number *N* (> 1) is specified, the topmost *N* FSMs are unioned.
`intersection` `intersect`	[binary] Compute the intersection (conjunction) of two FSMs (both must be acceptors). ***Note*: for best performance both operands should be (i)label sorted.**
`difference`	[binary] Compute the difference of two acceptors. The second operand must be deterministic and unweighted. Note that the operand on top of the stack will be the second operand of the operation. ***Note*: for best performance both operands should be label sorted.**
`compose [do-not-connect]` `composition [do-not-connect]`	[binary] Compute the composition of two FSMs. Note that the operand on top of the stack will be the second operand of the composition. For debugging purposes the option `do-not-connect` may be specified to prevent the final connection step. ***Note*: for best performance the first operand should be olabel sorted, and the second operand should be ilabel sorted.** See also: `compose`

`compose3`	[ternary] Compute the composition of three FSMs (the FSM on top of the stack being the last one). Currently, all three FSTs should not contain <phi> and <?>-transitions. Also, the outer FSMs are currently restricted to acceptors. The command is best suited for edit-distance computations where the inner FST represents a weighted edit-distance function. ***Note*: for best performance the first operand should be olabel sorted, and the third operand should be ilabel sorted.** See also: `edit-distance-fst`
`crossproduct` `product`	[binary] Create the crossproduct of two FSAs. Note that the operand on top of the stack will be the second component of the product.

Optimization and conversion

`rmepsilon [incremental]`	[unary] Remove ε-transitions. If `incremental` is given, an alternative algorithm (not based on distance computation) is used. This incremental algorithm is used by default in "p-subsequential" semirings not having the path property (like the string and unification semiring).
`determinize`	[unary] Determinize FSM (in case of certain FSTs and weighteds FSAs) this may lead to an endless loop. Note that weighted acceptors (not transducers) over certain "*p*-subsequential" semirings (like the string and unification semiring) admit *p*-subsequential determinization. This means that a finite number of weights may be associated with some final state.
`minimize`	[unary] Minimize (weighted) FSA
`optimize [ilabel|olabel] [encoded]`	[unary] Apply rmepsilon -> determinize -> minimize, encoded minimization in case of FSTs. If `ilabel` or `olabel` are specified the result of the optimization is sorted after the given criterion. If `encoded` is specified, weighted acceptors are also treated with encoded minimization.
`synchronize`	[unary] Try to synchronize an FST. May lead to an endless loop.
`epsnormalize`	[unary] Try to rearrange all ε-output labels of an FST after all non-ε-output labels. May lead to an endless loop.
`push weights [initial|final| residual-weights-final]`	[unary] Push the weights in FSM towards the initial or final state(s). Default is `initial`. If `residual-weights-final` is specified, the weight potential of the start state is multiplied with all final states' weights (only for commutative semirings).
`push labels [initial|final]`	[unary] Push the output labels in FST towards the initial or final state(s). Default is `initial`.
`sort ilabel|olabel|weight`	[unary] Sort the outgoing transitions of each state after the given sorting criterion.

`connect [renumber]`	[unary] Remove all useless states. If `renumber` is specified the states of the FSM will be renumbered afterwards to avoid gaps in the state numbering.
`compact`	[unary] Reduces memory requirements by reallocating state and transition tables.
`collect-weights`	[unary] Replaces all identically labelled transitions leaving a state and heaving the same destination state a single transition, where the weights are semiring-added.
`remove-weights` `neutralize-weights`	[unary] Replaces all weights by the neutral element of multiplication (defined by the semiring). The result will nevertheless be weighted.
`invert-weights`	[unary] Replaces all transition and final states weights by their multiplicative (left) inverses.
`convert` `list|table|matrix|compressed`	[unary] Converts the representation of the FSM on top of stack to another format. By the format specifier the type of the external file is determined: `list`: default read-write format `table`: efficient read-only format with fast access to the transitions of a state. Better memory footprint than `list`. The `matrix` and `compressed` formats are experimental and should not be used.
`save-as-cpp` *`classname`*	[unary] Converts the FSM on top of stack into C++ code and creates pair of files *classname.cpp* and *classname.h*. The FSM must be a deterministic acceptor over the string semiring; *classname* must be a valid C++ identifier.

Commands related to statistical modelling

Note: the commands for statistical language modelling assume that the symbol specification defines the symbols <s>, </s> and <alpha>. The first two of these are used for delimiting the sentences of a given corpus, while <alpha> is used for implementing back-off models.

`language-model N backoff|interpolation good-turing|witten-bell|abs-discount|kneser-ney|mod-kneser-ney`	[unary] Create a language model of order N (N ≥ 1). See section *Language Modeling*.
`ngram counter N [simultaneous]`	Create an FST which counts N-grams and put it on the stack. If `simultaneous` is given also all M-grams with M < N are counted. *Note*: For counting to be effective, the semiring must be non-idempotent. Examples are the probabilistic (real) and the log semiring.
`ngram concatenator N [backoff] [SIGMA]`	Create an FST which concatenates N-grams and put it on the stack. Specifying `backoff` results in a concatenator with failure transitions. If `backoff` is given the symbol specification must define the special symbol <alpha> . If SIGMA, a valid supertype in the symbol spec is defined, the concatenator is based on the subtypes of SIGMA
`ngram expander N`	To be documented
`ngram navigator N FAILSYM`	Create a special N-gram navigator WFSA for an N-gram back-off model. FAILSYM must denote the symbol used to navigate in a superordinate or subordinate distribution.
`ngram unfolder N`	To be documented
`print language-weight`	[unary] Computes the language weight of the FSA on top of stack, that is, the abstract sum of weights of strings accepted by the FSA.
`print entropy`	[unary] Print the entropy of the input FSA. Not yet.
`print relative-entropy`	[unary] Not yet.

Commands related to pattern matching

`suffix-fsm`	[unary] Create an FSA which accepts all suffixes of strings accepted by the argument acceptor
`failure-fsa FILE`	Creates a Aho-Corasick-style pattern matcher from the words in FILE.
`longest-match-fst DICTFILE [failure] [square-brackets| underscores]`	Creates a longest-match sequential string rewriting FSA from DICTFILE. If failure is specified the construction of the FSA is based on failure transitions. Otherwise, the method of Mihov and Schulz (Efficient Dictionary-Based Text Rewriting using Subsequential Transducers, 2005) is used. The FSAs resulting from the failure-method are generally much smaller than the complete automata created by the method of Mihov and Schulz. DICTFILE is a two-column, tab-separated text file with the input string in the first and the output string in the second column. See section *File Formats*. The last option handles the markup of the mapped pattern: `square-brackets` cause the mapped patterns to be enclosed in [ ], while `underscores` causes that spaces are replaced by underscores. Works only in the string semiring. Also supports UTF-8 mode, if the symbol specification is loaded with the utf-8 option.

Commands related to distance computations

`edit-distance-fst "<DELETE-COST>" "<INSERT-COST>" "<REPLACE-COST>" "<IDENTITY-COST>"`	Creates a weighted edit-distance transducer. In idempotent semirings (tropical, Viterbi), this is a 1-state transducer with *a*:ε, ε:*a*, *a*:*a* and *a*:*b*-transitions for each symbol $a,b \in \Sigma$. *a*:ε-transitions are weighted with `DELETE-COST`, ε:*a*-transitions with `INSERT-COST`, *a*:*b*-transitions with `REPLACE-COST` and *a*:*b*-transitions with `IDENTITY-COST`, respectively. In non-idempotent semirings (real, log), the result is a 2-state transducer representing a counting rational power series with looping *a*:ε, ε:*a*, *a*:*a* and *a*:*b*-transitions in both states (all weighted with SR::one()). Between the two states are *a*:ε, ε:*a* and *a*:*b*-transitions for each symbol $a,b \in \Sigma$, weighted as in the idempotent case. Note that *a*:*a*-transitions are not present. For optimal performance, the FST created by `edit-distance-fst` is input label-sorted and in table format. See also: `compose3, convert`

Test commands

`test symbol „s1“ „s2“ …`	Test whether s1, s2 … are symbols defined in the symbol specification.
`test acyclic`	[unary] Returns *true*, if the FSM on top of the stack is acyclic, otherwise *false*.
`test epsilon-free`	[unary] Returns *true*, if the FSM on top of the stack is epsilon-free, that is, has no epsilon (epsilon:epsilon) transitions, otherwise *false*.
`test equal-length`	[unary] Checks whether both tapes of the FSM on top of the stack are of equal length (meaningful only for transducers). Note: only a weak version of this property is checked, in particular, whether the FST contains ε:x or x:ε transitions. Even if an FST contains these kinds of transitions the underlying relation may be an equal length relation. To be sure apply first label pushing (see `push labels`) and then check for the equal-length property.
`test deterministic`	[unary] Checks whether the input FSM is deterministic (acceptor) or subsequential (transducer). *Note*: the function checks only for a weak version of determinism, that is, epsilon counts as a normal symbol. If a state has a single successor state which is reachable by an epsilon (epsilon:x) transition then this state would count as a deterministic one.
`test functional`	[unary] Not yet
`test ambiguous`	[unary] Not yet
`test twins-property`	[unary] Not yet

Semiring commands

`semiring`
`[tropical|tsr|`
`probabilistic|psr|`
`log|lsr|`
`string|ssr|`
`maxtimes|msr|viterbi|`
`maxplus|arctic|`
`unification|fsr|`
`tsr_x_tsr|`
`tsr_x_asr|`
`ssr_x_tsr|`
`ssr_x_psr|`
`ssr_x_lsr|`
`ssr_x_msr|`
`fsr_x_msr|`
`fsr_x_tsr]`
	Change the currently used semiring (see next subsection). If no argument is given, the current semiring is shown.
	Normally, the FSM stack must be empty, before changing the semiring is allowed. In the case of some of the numerical semirings (`tropical`, `probabilistic`, `log` and `viterbi`), the stack may be non-empty. In that case, the automata on the stack will be converted automatically to the requested semiring (transition and final state weights are translated accordingly). This can be usefully for language modelling purposes, where a WFSA over the probabilistic semiring is converted to the Viterbi semiring to compute best paths etc.
	
	Currently, the `semiring` command deletes all variables and substitutions defined with the `define` and `map` command, resp.
	
	Note that the actually available semirings depend on compilations options in the Makefile of *fsm2*.

All variables, encodings and substitutions are preserved if you change the semiring and later switch back to the original one.

Semirings

All FSM algebra operations are defined w.r.t. an abstract weight structure, a so called ***semiring***. The available *simple semirings* are[5]:

Name	Defined on	Operation applied to the weights along a path	Operation applied to multiple paths
`tropical` `tsr`	numbers	Addition	Minimum
`probabilistic` `psr`	numbers	Multiplication	Addition
`log` `lsr`	numbers	Addition	Log-Plus $-log\ (e^{-x} + e^{-y})$
`string` `ssr`	strings	Concatenation	Longest common prefix

`maxtimes`
`msr`
`viterbi`
	numbers	Multiplication	Maximum

`arctic` `maxplus`	numbers	Addition	Maximum
`unification` `fsr`	feature structures	Unification	Anti-unification (generalisation)

[5] The actually available semirings depend on options specified during the compilation process.

In addition, the following *complex semirings* are defined:

Name	Meaning
`tsr_x_tsr`	Ranked product of two tropical SRs *Note that is a ranked semiring product, that is, this semiring is idempotent and can be used in best path operations.*
`tsr_x_asr`	Ranked product of a tropical SR and an arctic SR
`ssr_x_tsr` `string_x_tropical`	Product of a string SR and a tropical SR (key-value semiring)
`ssr_x_psr`	Product of a string SR and a probabilistic SR (key-value semiring)
`ssr_x_lsr`	Product of string SR and log SR (key-value semiring)
`ssr_x_msr`	Product of string SR and Viterbi SR (key-value semiring)
`fsr_x_msr`	Product of unification SR and Viterbi SR (key-value semiring)
`fsr_x_tsr`	Product of unification SR and Tropical SR (key-value semiring)

The default semiring is the tropical semiring.
Semirings are changed with the `semiring` command followed by one of the mnemonics specified in the leftmost column in the two tables above.

A note on the string and unification semirings

Currently, not all operations are supported in the string/unification semiring (and the semirings having the string semiring as a component). This will change in the future.
Both string and unification semirings support operations (like determinization and minimization) on ***p*-subsequential** automata. That is, determinization of WFSMs over these semirings may result in deterministic WFSA where the final states may be associated with a *set of final weights*.

Semiring types

Currently, *fsm2* defines the semiring types explained in the following table:

Type	Members	Meaning

Simple semiring (-*p*-subsequential)	`tropical`
	`log`
	`probabilistic`
	`viterbi`
	`arctic`
		Simple semirings.
		Final weights were semiring-added

Simple semiring (+*p*-subsequential)	`unification` `string`	Simple semiring which admit multiple final weights.

Ranked tuple semiring	`tsr_x_tsr`
	`tsr_x_tsr_x_tsr`
	`tsr_x_asr`
		Lexicographic, idempotent tuple semirings with the path property, suitable for best path search

Key-value semiring	`ssr_x_tsr`
	`ssr_x_lsr`
	`ssr_x_psr`
	`ssr_x_msr`
	`fsr_x_msr`
	`fsr_x_tsr`
		p-subsequential semirings admitting multiple final state outputs. The ⊕-operation applies ⊕ to both key and value semirings. When determinizing, the values of final outputs with the same key are additively combined. When the keys are different, the key-values pairs are added to the final output set.

The unification (feature structure) semiring

Basically, an (untyped) feature structure is a directed, acyclic graph where edges are labelled with *features* and final states with *atomic feature values*.

The following table summarizes the (Prolog-style) syntax of feature structures in *fsm2*:

Syntax rule	
FS	→ `[` *FSLIST* `]` | `[]` | **`$top$`**
FSLIST	→ *FEAT* `:` *VALUE*
FSLIST	→ *FEAT* `:` *VALUE* `,` *FSLIST*
FEAT	→ `[a-z] [A-Za-z0-9]*`
VALUE	→ *OPT_COREF* `(`*ATOM* | *FS*`)`
VALUE	→ *VAR*
ATOM	→ `[a-z0-9] [A-Za-z0-9_]*` | `+` | `-` | `*`
VAR	→ `[_A-Z] [A-Za-z0-9]*`
OPT_COREF	→ *VAR* `=` | ε

The next table lists some examples for feature structures in *fsm2*:

Example	Comment
`[]`	*The empty feature structure (bottom, ⊥)*
`[num:pl,case:nom]`	
`[f:X,g:X]`	*A feature structure where the features **f** and **g** co-refer to the same value (which is bottom, the most general feature structure)*
`[subj:X=[agr:[pers:3, num:pl]],pred:X]`	*A feature structure where the features **subj** and **pred** co-refer to the same value which is a complex feature structure*
`$top$`	*The inconsistent feature structure (signalling unification failure)*

Example: German lexicon with some forms of the verb *werfen* (*to throw*)

```
geworfen    <[vform:part2]>
wirf        <[vform:fin,mood:imp,num:sg]>
werft       <[vform:fin,mood:imp,num:pl]>
werfe       <[pers:1,num:sg,tns:pres,mood:ind,vform:fin]>
wirfst      <[pers:2,num:sg,tns:pres,mood:ind,vform:fin]>
wirft       <[pers:3,num:sg,tns:pres,mood:ind,vform:fin]>
werfen      <[pers:1,num:pl,tns:pres,mood:ind,vform:fin]>
werft       <[pers:2,num:pl,tns:pres,mood:ind,vform:fin]>
werfen      <[pers:3,num:pl,tns:pres,mood:ind,vform:fin]>
```

Using macros

A *fsm2* macro can be understood as a subprogram with parameters. Macros can be defined in scripts or interactively.

Commands related to macros

`macro MACRONAME( PARAM`$_1$ `... PARAM`$_N$ `)`	Starts the definition of macro `MACRONAME`. In interactive mode, the interpreter prompt changes to `MACRONAME`. `MACRONAME` may consist out of upper and lowercase letters, digits, underscore and hyphen characters. Macro names are case-sensitive.
`endmacro`	Finishes the definition of the current macro.
`call MACRONAME( PARAM`$_1$ `... PARAM`$_N$ `)`	Calls the macro specified by `MACRONAME`. Each macro call is executed within its own subinterpreter. If a macro leaves its FSM stack non-empty, the top element of the subinterpreter's stack will be automatically pushed on the stack of the calling interpreter.
`foreach ( VAR in FILE )` `call MACRONAME( PARAM`$_1$ `... PARAM`$_N$ `)`	Reads FILE line by line and call macro `MACRONAME` for each line.
`print macros`	Prints all currently defined macros.
`print macro MACRONAME`	Prints the definition of the macro called `MACRONAME`.
`clear macros`	Clear all macro definitions.

The basic structure of a macro is the following:

`macro` `MACRONAME ( PARAM`$_1$ `... PARAM`$_N$ `)`
command$_1$
…
command$_k$
`endmacro`

Macro example

```
macro test(FN,RE)
  load symspec %FN%
  regex "%RE%"
endmacro
```

Formal macro parameters used in the macro's body are enclosed with %.
Macros are called with the **`call`** command:

Macro example (con'd)

```
call test(sigma,ab)
```

The `foreach` command can be helpful to apply a sequence of commands to every line of a given file, for example, for testing purposes.

Example: Using foreach

```
macro print_words(RE)
  regex "%RE%"
  print words
  echoln
  pop
endmacro

# Assume that "text.txt" contains the following entries, one in each line:
# abc, ca|ab, cde

foreach (R in "test.txt") call print_words(%R%)
```

Some notes on macros

- Macros can call other macros (but a macro cannot call itself).
- A macro must have been defined before it can be called.
- There are two types of macro parameters:
  1. *String parameters*: file names, constants, regular expressions, command parameters (but not keywords of commands or subcommands). Each occurrence of a formal string parameter within a macro definition is replaced by the actual string parameter. Although you can pass regex or filename parameters as such to a macro, it is recommended that you enclose them in double quotes, especially if they contain special symbols like brackets.
  2. *FSM parameters*: these are given by a local or global FSM variable which must not be enclosed in double quotes. FSM parameters are passed to the macro by value.
- Variables defined with the `define` command inside a macro are local to this macro
- FSM variables outside the scope of a macro definition are global and can be used inside a macro.
- If there are a local and a global variable with the same name, the local variable is preferred.
- All macros operate on different FSM stacks. If you want to transfer information to a called macro, this can be solely done by using macro parameters or global variables (**this changed from prior versions of *fsm2* where a macro had access to the calling macro's stack**).
- To use the top FSM of the caller's FSM stack as a macro argument, *fsm2* defines the special variable `%TOS%` (top of stack).
- When a macro call finishes, it is checked whether the stack of the called macro is non-empty. If this is the case, the topmost FSM on the called macro's stack is pushed on the stack of the calling macro. In that way, a macro can return FSMs to the caller.
- Macro definitions cannot be nested.

Example

```
macro replace(ALPHA,BETA,GAMMA,DELTA)
  regex “%ALPHA% -> %BETA% / %GAMMA% _ %DELTA%”
endmacro

macro replace_test(FN,ALPHA,BETA)
  load symspec %FN%
  # Define two local variables for the left and right context
  define LC c
  define RC d
  # The replace macro is called with two string parameters
  # (ALPHA and BETA) and two FSM parameters (LC and RC)
  call replace(%ALPHA%, %BETA%, %LC%, %RC%)
endmacro

# Call replace_test. The result will be placed on the stack
call replace_test(“sigma.sym”, ”ab|cd”, ”x”)
```

Using variables

Variables can be defined with the `define` command:

Variable example

```
[0]> load symspec sigma
Symbol specification sigma.sym (6 user symbols, 4 supertypes, 0
categories) loaded in 0s
[0]> regex "ab"
[acceptor, 3 states, 1 final states, 2 transitions]
[1]> define var1
Variable var1 defined: [acceptor, 3 states, 1 final states, 2 transitions]
[1]> define var2 "ba"
Variable var2 defined: [acceptor, 3 states, 1 final states, 2 transitions]
[1]> print vars
var1    [acceptor, 3 states, 1 final states, 2 transitions]
var2    [acceptor, 3 states, 1 final states, 2 transitions]
[1]> regex "%var1%|%var2%"
[acceptor, 5 states, 2 final states, 4 transitions]
[2]> print words
ab
ba
```

Defined variables (enclosed in %) may occur everywhere in a regular expression. The %s must be omitted after the commands `push stack`, `define` and `undefine`.

Commands related to variables, encodings and substitutions

`define VAR ["REGEX"]`	Assign the FSM denoted by `REGEX` to variable `VAR`. If `REGEX` is not specified, `VAR` will be mapped to the FSM on top of the stack.
`undefine VAR`	Undefine variable `VAR`.
`push stack VAR`	Push the FSM assigned to `VAR` on the stack.
`print vars`	Print all currently defined variables.
`clear vars`	Delete all currently defined variables.
`clear encodings`	Clears all currently defined encodings. See also: `encode`, `decode`
`clear substitutions`	Clears all currently defined substitutions. See also: `map`, `substitute`

Language Modeling

fsm2 supports the construction of different kinds of *N*-gram language models over different semirings. A language model is a probability distribution over Σ* such that the individual string probabilities sum up to one. In terms of finite automata, a language model is a cyclic, deterministic and robust WFSA over a real-valued semiring which assigns a (log)-probability to every input string over some given alphabet Σ*. Suitable choices for semirings are the probabilistic, logarithmic, Viterbi or tropical semiring.
In principle, language models can be divided into backoff and interpolation models (for a clarification of these notions, refer to Chen & Goodman (1998): *An Empirical Study of Smoothing Techniques for Language Modeling*).
Language modeling is supported in *fsm2* by the `language-model` command:

`language-model` command syntax
`language-model N backoff|interpolation good-turing|witten-bell|abs-discount|kneser-ney|mod-kneser-ney`

N is the *N*-gram parameter (usually 2 or 3) and `backoff` and `interpolation` define the combination method of the different subdistributions. Backoff model use failure transitions to continue processing in the next subdistribution in case of failure.
The last parameter determines the smoothing method (please refer to Chen & Goodman for an explanation of these methods.

For using the `language-model` command, the symbol specification must define the two supertypes `SentenceBeginDelimiter` and `SentenceEndDelimiter`, for example:

Example: Delimiter definition	
`SentenceBeginDelimiter`	`<s>`
`SentenceEndDelimiter`	`</s>`

Depending on *N*, the corpus which forms the base for the language model must be prefixed with *N*-1 instances of `SentenceBeginDelimiter` and suffixed with a single instance of `SentenceEndDelimiter`. When applying the language-model to some input sentence, it must be enclosed in sentence delimiters in the same way.

Example: Construction of a toy bigram model

```
semiring psr
regex "[<s>](babbaaabbbabbbabbabbabba|abbbabb|aaababbbabbbabaaa)[</s>]"
language-model 2 backoff witten-bell
apply "[<s>]baba[</s>]"
```

Class-based language models

Class-based language models combine a *lexical distribution* and a *class distribution*.
We assume that each input word $w \in Word$ is mapped to a number of classes $\subseteq Classes$, for example part-of-speech categories. A lexical distribution L may then represent the conditional distribution $P(w|c)$ for all $w \in Word$ and $c \in Classes$.
The class distribution C is a language model based on *Classes* as alphabet.
Since typically L maps an input symbol to several classes with different probabilities (for example in case of homography), the composition $L \circ C$ is a nondeterministic transducer which may allow a number of successful paths for some input sentence. To choose a (the) best one for some given input, we need an ordering amongst the weights assigned to each path (see below).
The following example shows the construction of a conditional distribution from a tagged corpus (WORD denotes the supertype for input words, POS the supertype for POS tags):

Example: Creation of a lexical distribution for bigram class models

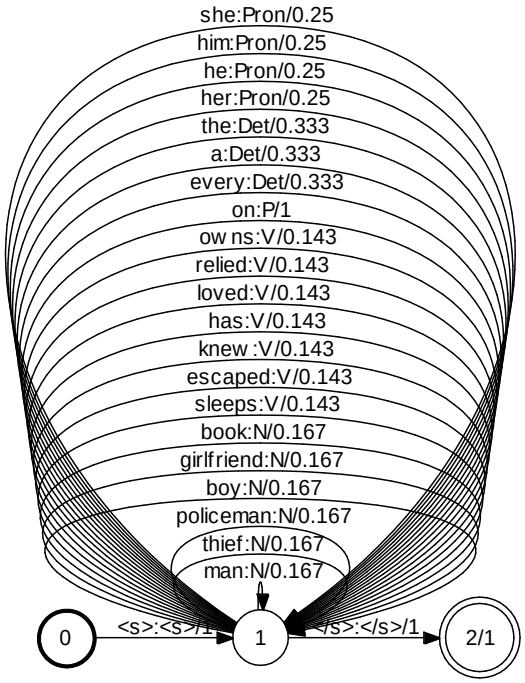
```
semiring psr
load acyclic-lexicon tagged_corpus
define word_pos_loop "((([<sigma>])*)/d/m : [])/m"
# Create a counting series for WORD-POS-pairs and apply it to the corpus
regex "(%word_pos_loop%) (%WORD%%POS%) (%word_pos_loop%)"
compose
project 2
optimize
reverse
optimize
# Create a conditional distribution P(WORD|POS)
probabilize 2 conditional
# Create a cyclic transducer mapping words to their POS tags
regex "(%POS%:[])%WORD%"
compose
invert
regex "(%POS%)(%WORD%:[])"
compose
synchronize
closure
# Enclose the resulting FST within sentence delimiter symbols
regex "[SentenceEndDelimiter]"
concat
regex "[SentenceBeginDelimiter]"
flip
concat
optimize olabel
```

The input lexicon `tagged_corpus.lex` may look like this (each word is followed by its corresponding POS tag):

Example: Input lexicon for the lexical distribution

```
[a][Det] [man][N] [sleeps][V]
[the][Det] [thief][N] [escaped][V] [the][Det] [policeman][N]
[every][Det] [boy][N] [has][V] [a][Det] [girlfriend][N]
[the][Det] [policeman][N] [owns][V] [a][Det] [book][N]
[every][Det] [man][N] [owns] [V][a] [Det] [book][N]
[he][Pron] [knew][V] [the][Det] [thief][N]
[he][Pron] [loved][V] [her][Pron]
[she][Pron] [relied][V] [on][P] [him][Pron]
```

The resulting transducer looks like this:



The second component, the class-based language model *C*, is created only from the tags taken from this lexicon (again enclosed within delimiter symbols).

The construction of language models is currently limited to the probabilistic semiring. After construction, the resulting FSM may be converted to other numerical semirings with the `semiring` command. This is necessary for parsing issues since determining the best path for some input requires an idempotent semiring like the Viterbi or tropical semiring. Additionally, it is useful to reinterpret the probabilities within the log-space (as negated log-probabilities) to ensure numerical stability (the tropical and log semirings share this property). A semiring which is both stable and idempotent is the tropical semiring.

FSM<2.0> Regular Expression Syntax

Note: it is advisable to enclose the regular expressions in double quotes.

Example: `regex "[NOUN Case=nom]"`

Literal double quotes inside regular expressions must be prefixed with `\`.

Basic regular expressions	**Examples**
a single symbol	`A B C a b c ä`

a quoted symbol	`[Case] [$] \*`
	Quoting (with []) is generally necessary in case of sequences with length > 1 or the usage of predefined operators symbols in a literal way.
	The following symbols must be quoted (either in [] or with \):
	`* + | ? : & - @ ! ^ ~ # \ / % $`
	`) (  ] [ } { >  < _ . =`

a category with or without features
(the order of the features is irrelevant; the order of the feature values in the resulting FSM is determined by the definition of the category)
	`[NOUN Case=nom Gender=fem]`
	`[NOUN Gender=fem Case=!nom]`
	`[NOUN Gender=masc:fem]`
	`[NOUN Gender=masc|fem]`
	`[NOUN Gender=masc:fem|neut:fem]`
	`[VERB]`
	`NOUN` and `VERB` are defined as categories
	In general, the feature-value syntax is as follows:
	feat-val → *feature* = !*value*
	feat-val → *feature* = *value-list*
	value-list → v_1 v_2 ... v_k
	v_i → *value* | *value* : *value*

a numerical symbol of the form `[#decimal]` to access a symbol by its unique symbol number	`[#99]`
a cost in < > The exact format of the cost within < > depends on the chosen semiring: • Numerical semirings (`psr`, `tsr`, `lsr`, `msr`, asr): an integer or float • String semiring (`ssr`): a string in single quotes (single quotes within the string must be prefixed with \) • Unification semiring (`fsr`): a feature structure in [] or `$top$` • Tuple semirings: the tuple components are stated in round brackets ( ) and are separated with commas	`<1.0>` [tropical SR] `<'abb'>,` `<'Moe\'s Bar'>` [string SR] `<[f:a,g:b]>` [unification SR] `<('xy',5)>` [(string,tropical) SR] `<(2.0,5.4)>` [(tropical, tropical) SR] `<(2,3,5)>` [(trop, trop,trop) SR] `<[f:a,g:b],0.3>` [(unification,Viterbi) SR]

Basic regular expressions	Examples
`[<epsilon>], [<eps>], [<e>], []` (the empty string)	
`[<sigma>]` (all alphabet symbols)	
`[<category>]` (all categories)	
`[<category_features>]` (all category features)	
`%varname%`	If `%varname%` is a defined variable (see `define` command), it is replaced by the associated FSM. Variable names may consist out of the following symbols: A-Z, a-z, 0-9, _, and -

Note that weight strings (enclosed in < and >) may not contain < or >, not even masked.

Recursive regular expressions	
Prefix operators	
`! RE`	complement of RE. RE must denote an unweighted acceptor.
`^ RE`	inversion of RE (exchange of input and output labels). RE must denote a transducer.
`$ RE`	contains operator (equivalent to `Σ* RE Σ*`)
Infix operators	
RE_1 RE_2	Operatorless concatenation
RE_1 `|` RE_2	Disjunction
RE_1 `-` RE_2	Difference
RE_1 `:` RE_2	Cross product
RE_1 `&` RE_2	Intersection
RE_1 `@` RE_2	Composition
RE_1 `/iff` RE_2	*Iff-suffix-then-prefix* operator. Denotes the language where every instance of RE_2 is exactly preceded by an instance of RE_1. RE_1 and RE_2 must denote unweighted acceptors; the result will be a deterministic acceptor.
`RE /i [TYPE]`	Ignore in RE the terminal types in TYPE (TYPE must be a type (super type or maximal subtype) in brackets)
Postfix operators	
`RE *`	star closure of RE
`RE +`	plus closure of RE
`RE ?`	optional version of RE
`RE /1`	first projection of RE (RE must denote a transducer)
`RE /2`	second projection of RE (RE must denote a transducer)
`RE /r`	reversal of RE
`RE /b`	the best path in RE (used semiring must be idempotent)
`RE /pw`	push weights in RE towards initial state (RE must denote a weighted FSM)
`RE /pl`	push labels in RE towards initial state (RE must denote a FST, may change its topology)
`RE /e`	remove epsilon transitions from RE
`RE /d`	determinize FSM for RE (may loop for certain FSTs)
`RE /m`	minimize FSM for RE (only if RE denotes an acceptor, otherwise no effect)

Rule operators	
`alpha -> beta`	Obligatory replacement of an instance of `alpha` by an instance of `beta.` `alpha` must denote an unweighted acceptor, `beta` may be weighted.
`alpha (->) beta`	Optional replacement of an instance of `alpha` by an instance of `beta`
`alpha -> beta / gamma _ delta`	Obligatory replacement of an instance of `alpha` by an instance of `beta` if `alpha` is preceded by an instance of `gamma` and followed by an instance of `delta`.
`alpha (->) beta / gamma _ delta`	Optional replacement of an instance of `alpha` by an instance of `beta` if `alpha` is preceded by an instance of `gamma` and followed by an instance of `delta`.
`alpha -> left … right`	Obligatory bracketing of `alpha` with `left` and `right`
`alpha (->) left … right`	Optional bracketing of `alpha` with `left` and `right`
`alpha @-> beta`	Longest match replacement of `alpha` by `beta`
`alpha @-> left … right`	Longest match bracketing of `alpha` with `left` and `right`
`supertype1 => supertype2`	Parallel replacement: Obligatory replacement of each direct subtype of `supertype1` with the corresponding direct subtype of `supertype2`. The order is determined by the definition of the two super types in the symbol specification. Both super types must have the same number of subtypes.
`supertype1 (=>) supertype2`	Parallel replacement: Optional replacement of each direct subtype of `supertype1` with the corresponding direct subtype of `supertype2`. The order is determined by the definition of the two super types in the symbol specification. Both super types must have the same number of subtypes.
`supertype1 => supertype2 WEIGHT` `supertype1 (=>) supertype2 WEIGHT`	Same as above, but each symbol replacement is associated with WEIGHT

Regular expression precedence table

Operator	Type	Meaning	Associativity
`@ .o.`	infix	Composition	left
`-->`	infix	CFG operator	nonassoc
`-> (->)`	infix	Replacement rule	nonassoc
`=> (=>)`	infix	Parallel replacement rule	nonassoc
`@->`	infix	Longest match rule	nonassoc
`/`	infix	Context introduction	nonassoc
`|`	infix	Union	left
`&`	infix	Intersection	left
`-`	infix	Difference	left
`:`	infix	Cross product	left
`!`	prefix	Complement	right
`^`	prefix	Inversion	right
`/r`	postfix	Reversal	left
`+ *`	postfix	Reflexive & transitive Closure	left
`?`	postfix	Optionality	left
`/1 /2`	postfix	1st, 2nd projection	left
`/b`	postfix	Best path	left
`/i`	infix	Ignore symbols	left
`/pw`	postfix	Push weights	left
`/pl`	postfix	Push labels	left
`/e`	postfix	Remove epsilon	left
`/d`	postfix	Determinization	left
`/m`	postfix	Minimization	left

Precedences go from low to high.

Lexical element	
Category name	
Feature names	
Feature value	May consist out of any symbol, but of the following: `= ! | : [ ] #` and white space
Supertype / subtype	

File formats

A word on encodings

...

Symbol specifications

This section explains the format of symbol specification file which is the main argument of the `load symspec` command.
The default file extension is .sym.

Basic format of a symbol specification

A symbol specification file consists of a collection of entries.
An entry can have two possible forms:

`Supertype  Subtype`$_1$ `Subtype`$_1$ `... Subtype`$_k$
`Category:` `Cat Subtype`$_1$ `Subtype`$_1$ `... Subtype`$_k$

Example	
`Case`	`acc dat gen nom`
`Gender`	`fem masc neut`
`Number`	`sg pl`
`Person`	`1 2 3`
`Person_or_Number`	`Person Number`
`Category: NOUN`	`Gender Number Case`
`Category: VERB`	`Person Number`

A subtype occurring on the right hand side of a supertype definition can be the supertype of further subtypes. The inheritance hierarchy must form an acyclic directed graph.
The types must be ordered in such a way that all types mentioned in the right hand side of a super type definition must be already defined (that is, the type definitions must be written in reverse topological order of the underlying inheritance graph).
`Category:` is a case-sensitive keyword. Category and symbol names are also case-sensitive.
A category name (like `NOUN` and `VERB`) in the example above is also called a *category marker*.
The mapping from types and category names to symbols numbers is done through a depth-first traversal of the graph. A category will be prefixed with an underscore and will be assigned a unique symbol number. For example the category name `NOUN` will be transformed to `_NOUN` and assigned a unique symbol number.
Only maximal subtypes (that is, types which do not have further subtypes and thus form the leaves of the inheritance hierarchy) and category markers are assigned a symbol number to. On the other hand, supertypes (as for example `Person` above) are expanded to the finite disjunction of their (transitive) subtypes. They are thus not represented directly in an automaton compiled from some regular expression.

Special types and symbols

Besides the symbols mentioned in the specification file there are a number of predefined special symbols:

`<epsilon>`	The symbol representing the empty string ε (always mapped to 0). Note that there is a second implicit ε-symbol labelling a self-loop entailed by the reflexive case of the δ^* transition function.
`<?>`	The *default* (also called *otherwise*) symbol (mapped to -3) always consumes a symbol in the other operand. If a state *q* considered while intersecting or composing two FSMs has an outgoing transition labelled with `<?>`, this transition is taken for every unmatched symbol in the other operands. *Examples*: • If FSM1 is **"abc"** and FSM2 is **"a[<?>]c"**, their intersection will be **"abc"**. • If FSM1 is **"abc"** and FSM2 is **"(a:x|[<?>]:y)*"**, their composition will be **"abc:xyy"**. • If FSM1 is **"abc"** and FSM2 is **"(a:x|[<?>])*"**, their composition will be **"abc:xbc"**. • If FSM1 is **"a:[<?>]"** and FSM2 is **"[<?>]:b"**, their composition will be **"a:b"**. As can be seen from the second to last example, if an operand contains an ID(`<?>`) transition, `<?>` acts like a variable: if `<?>` matches on the upper tape *a* symbol a, the output tape after composition will contain also *a*.
`<phi>`	The *conditional epsilon* symbol (always mapped to -2) behaves like `<epsilon>`, but depends on the absence of other matching transitions. It is also called the *failure transition* (and for example used in the result of the `failure-fsa` command). If a state *q* considered while intersecting or composing two FSMs has an outgoing transition labelled with `<phi>`, this transition is taken, if no other normal transition can be used. *Examples*: • If FSM1 is **"ab"** and FSM2 is **"[<phi>]ab"**, their intersection will be **"[<epsilon>]ab"** = **"ab"** • If FSM1 is **"a"** and FSM2 is **"([<phi>]:x)a"**, their composition will be **"a:xa"**

`<bos>`	matches the beginning of an FSM Note that <bos> is a symbol which may only occur in the *left* context of a replacement rule. It then merely controls the rule compilation and disappears afterwards. *Example*: `[Lowercase] => [Uppercase] / [<bos>] _` replaces every lowercase letter by the corresponding uppercase letter at the beginning of an input.
`<eos>`	matches the end of an FSM Note that <eos> is a symbol which may only occur in the *right* context of a replacement rule. It then merely controls the rule compilation and disappears afterwards. *Example*: `[] -> [$] / _ [<eos>]` inserts a dollar sign at the end of each input string
`<sigma>`	`<sigma>` , being the topmost type, is expanded to all other terminal types (but not the a before mentioned special symbols)
`<category>`	is expanded to all defined categories (introduced with the `Category:` keyword)
`<category_features>`	is expanded to all terminal symbols (that is category markers like `_NOUN` and feature values) of all defined categories.

As can be seen from the examples, <phi> and <?> will never be present in the result of the composition/intersection.

Some restrictions apply to <phi> and <?>

- There can be at most one <?> or <phi> transition at any state.
- A <phi> transition may currently not contain a regular symbol on the input or output side of the transition. That is, transitions like "`[<phi>]:a`" or "`a:[<phi>]`" are ruled out.
- If both operands have corresponding states with outgoing <?> or <phi> transitions, both symbols are treated as ordinary symbols.
- If a state has both <?> and <phi> transitions, <?> will be given priority, regardless whether <?> matches a transition or not.

The following hierarchy describes the semantics and interplay of <phi> and <?>:

$$ TODO

Special supertypes

In addition, the user may define special supertypes for parametrizing tasks like language modelling or grammar compilation:

Supertype	Command	Description
`SentenceBeginDelimiter`	`language-model`	Defines the delimiter symbol for marking the beginning of sentences in language modelling tasks. The supertype `SentenceBeginDelimiter` must only have a single subtype, for example: `SentenceBeginDelimiter <s>`
`SentenceEndDelimiter`	`language-model`	Defines the delimiter symbol for marking the beginning of sentences in language modelling tasks. The supertype `SentenceEndDelimiter` must only have a single subtype, for example: `SentenceEndDelimiter </s>`

`NONTERMINAL`
`LEFTBRACKET`
`RIGHTBRACKET`
	`load grammar`	When using the `create-bracket-fst` option of load grammar, the symbol specification must define three super types which must have the same number of direct subtypes. `NONTERMINAL` defines all the nonterminals used in the grammar. The i^{th} subtype of `LEFTBRACKET` (`RIGHTBRACKET`) defines the left (right) bracket of the i^{th} subtype of `NONTERMINAL`. An example:
		
		`NONTERMINAL S NP VP`
		`LEFTBRACKET (S (NP (VP`
		`RIGHTBRACKET S) NP) VP)`

Special terminal symbols

`$unknown$`	If a symbol specification defines this special symbol (for example, as a subtype of some user-defined supertype), all undefined terminal types will be mapped to `$unknown$`. Thus, "undefined symbol" warnings will be avoided and the processing will become more robust. Note that this currently affects only the commands `regex` and `define`.

Other features

(in column 1) may be used to add comments.

Moreover you can recursively include further symbol files through the use of the `#include` statement:

Example

```
# Include a second symbol file
#include “symspec2.sym”
```

General lexicon files

General lexicon files are disjunctions of arbitrary regular expressions

Comments start with a # symbol (at any column)

Example

```
# Lexicon containing some English noun stems
# The symbol signature contains the following:
# Letter    a b c d e f g h I j k l m n o p q r s t u v w x y z
# Number    sg pl
# StemType  normal special
# Category: NSTEM Number StemType

dog               [NSTEM StemType=normal]
fox               [NSTEM StemType=normal]
box               [NSTEM StemType=normal]
piano             [NSTEM StemType=normal]
table             [NSTEM StemType=normal]
sheep             [NSTEM StemType=special]
butterfly         [NSTEM StemType=normal]
(goose : goose)   [NSTEM Number=sg StemType=special]
(goose : geese)   [NSTEM Number=pl StemType=special]
(mouse : mouse)   [NSTEM Number=sg StemType=special]
(mouse : mice)    [NSTEM Number=pl StemType=special]
```

Lexicons are loaded with the `load lexicon` command.

The default file extension is `.lex`.

Currently, there is no file inclusion mechanism.

"Acyclic" lexicon files

"Acyclic" lexicon files – they are called acyclic because the resulting FSM must be acyclic – are disjunctions of restricted regular expressions. Currently, these regular expressions may only use (invisible) concatenation. Lexicons may denote transducers – in that case an entry may contain a colon (:) somewhere in the line. Round brackets for grouping are currently not supported. Feature-value syntax is supported, but currently limited to the simplest form *feat=atomic-value* (so, no negation or disjunction operators are supported). Underspecification is supported but may lead to memory problems since the compiler expands the underspecified entry to disjunctive normal-form. Semiring weights (in < >) may occur everywhere in the line

Comments start with a # symbol (at any column)

Example: German indefinite determiners

```
ein : eine [ARTINDEF Number=sg Case=acc Gender=neut]
ein : eine [ARTINDEF Number=sg Case=nom Gender=masc]
ein : eine [ARTINDEF Number=sg Case=nom Gender=neut]
eine : eine [ARTINDEF Number=sg Case=acc Gender=fem]
eine : eine [ARTINDEF Number=sg Case=nom Gender=fem]
einem : eine [ARTINDEF Number=sg Case=dat Gender=masc]
einem : eine [ARTINDEF Number=sg Case=dat Gender=neut]
einen : eine [ARTINDEF Number=sg Case=acc Gender=masc]
einer : eine [ARTINDEF Number=sg Case=dat Gender=fem]
einer : eine [ARTINDEF Number=sg Case=gen Gender=fem]
eines : eine [ARTINDEF Number=sg Case=gen Gender=masc]
eines : eine [ARTINDEF Number=sg Case=gen Gender=neut]
```

Lexicons are loaded with the `load acyclic-lexicon` command. This command uses an efficient incremental compiler which creates minimal FSMS. The input file needs not to be sorted.

The default file extension is `.lex`.

Currently, there is no file inclusion mechanism.

Context rule files

Context rule files contain replacement rules of various types.
Lines may contain the special keywords `optional` and `obligatory` which control the application mode of the rules. The default is `obligatory`. The scope of these keywords reaches to the next restatement of one of them.
Rules are composed in the order in that they appear in the text file.
Comments start with a # symbol.

Examples

```
# Suppose the underlying symbol spec contains:
# Uppercase A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
# Lowercase a b c d e f g h I j k l m n o p q r s t u v w x y z
# Letter Uppercase Lowercase
# Gender masc fem neut
# Case nom acc
# Number sg pl
# Category: NSTEM Gender
# Category: NINFL Number Case
# Bracket   ( )

Obligatory
[Letter]+ -> [<epsilon>] / [NSTEM] _ [NINFL]
[NSTEM] -> [<epsilon>]
[_NINFL] -> [_NOUN]

# The next rules are interpreted optionally
optional
d -> z
e -> v|w
# Replace each uppercase with its corresponding lowercase letter
[Uppercase] => [Lowercase]
# Longest-match replacement
a(b*) @-> x|y
# Longest-match bracketing
a(b*) @-> [(] ... [)]
# Bracketing
a(b*) -> [(] ... [)]
```

Context rule files are loaded with the command `load contextrules`.
The default file extension is `.rules`.
Note that context rules are in general computationally expensive, so compiling a rule file with many rules can take a long time.

Grammar rules files

A grammar rule file consists of quasi context-free rules. The general format is:

```
LHS --> REGEX
```

Comments start with a # symbol (not necessary in column 1).

Example: simple probabilistic CFG (semiring = probabilistic)

```
# Simple PCFG
# (all probabilities for a specific nonterminal sum up to 1.0)

# Phrase structure
[S]    --> [NP] [VP] <1.0>
[NP]   --> [Det] [N] <0.7>
[NP]   --> [N] <0.3>
[VP]   --> [V] [NP] <0.5>
[VP]   --> [V] [NP] [NP] <0.35>
[VP]   --> [V] [S] <0.15>

# Lexicon
[N]    --> [children] <1.0>
[V]    --> [sleep] <0.3>
[V]    --> [know] <0.4>
[V]    --> [sleeps] <0.3>
[Det]  --> [the] <1.0>
```

Grammar rules may be split into several lines when using `\` as the last symbol in a line (which is not the last line):

Example: Grammar rule split into several lines

```
[VERB] -->  [VERBPREFIX]? \
            [VERBROOT] \
            [VERBSUFFIX]
```

Grammar files are loaded with the `load grammar` command. The default file extension is `.grm`.

A multiline rule can be commented out by adding a # at the beginning of the first line.

The right-hand side of a grammar rule may also specify a transduction.

Subgrammars may be included with the `#include "FILE"` statement.

There are some restrictions on the rules (of course, context-free languages can't be recognized by finite-state automata)[6].

[6] The technical restriction is the following: Every strongly-connected component of the grammar's dependency graph must be either left-linear or right-linear.

Grammar approximation

Grammars not fulfilling the technical requirement stated above can nevertheless be compiled approximately (with the `approximate` keyword). After approximation, the current directory will contain a file *grammar_file_name.approx* which contains the approximated grammar.

Dictionary rewriting files

The input file of the `longest-match-fst` command is a two-column, tab-separated text file with the input string in the first and the output string in the second column.

Example: Input file of the `longest-match-fst` command

```
his	1
her	2
she	3
he	4
us	5
```

If the second column is empty, the output string is equal to the input string.

Textual FSM files

fsm2 supports two types of text files to store FSMs: the AT&T format and an XML format. Files in one of these formats are stored and loaded with the `print fsm` and `load fsm` commands, resp. The expected file type in these operations is controlled with the `text-format` command.

AT&T format

The AT&T format is a simple tab-separated format in which the lines define the transitions of an FSM. They can be of the following forms:

1	2	3	4	5
`Source-state`	`Dest-state`	`Input-label`	`Output-label`	`[Weight]`
`Source-state`	`Dest-state`	`Label`	`[Weight]`	
`State`	`[Weight]`			

The first line describes the transitions of a (weighted) transducer, the second that one of a (weighted) acceptor. Final states are represented by a state and an optional weight. In case of multiple final weights, these lines may occur repeatedly. The source state of the first transition in the file is used as the start state.
States are positive numbers, labels are either numbers (with 0 denoting ε) or symbols, depending on the status of the `use-symbol-names` switch. Weights are represented in the genuine weight syntax (without < and >), so for example, strings are enclosed in ' etc.
Note that the AT&T format is underspecified with respect to the transducer property. Therefore, the user has to add the `transducer` option to the `load fsm` command when loading a transducer.

XML format

Examples

Compiling regular expressions

```
FSM<2.0> interactive interpreter v1.0.0 (tropical semiring)
Type 'help' for help on commands

[0]> load symspec sigma
Symbol specification sigma.sym (8 user symbols, 2 supertypes, 0 categories)
loaded in 0.078s
[0]> regex "abc|da|ac|cc|dbac|ba"
Regex compiled in 0s
[acceptor, 16 states, 6 final states, 15 transitions]
[1]> print words
abc
ac
ba
cc
da
dbac
```

Rule application

```
FSM<2.0> interactive interpreter v0.9.9 (tropical semiring)
Type 'help' for help on commands

[0]> load symspec sigma
Symbol specification sigma.sym (8 user symbols, 2 supertypes, 0 categories)
loaded in 0.078s
[0]> regex "xabcdx"
Regex compiled in 0s
[acceptor, 7 states, 1 final states, 6 transitions]
[1]> regex "ab|bc -> x"
Regex compiled in 0.016s
[transducer, 5 states, 3 final states, 30 transitions]
[2]> compose
Intersected/composed FSMs in 0s
[transducer, 9 states, 1 final states, 9 transitions]
[1]> project 2
FSM projected in 0s
[acceptor, 9 states, 1 final states, 9 transitions]
[1]> print words
xaxdx
xxcdx
```

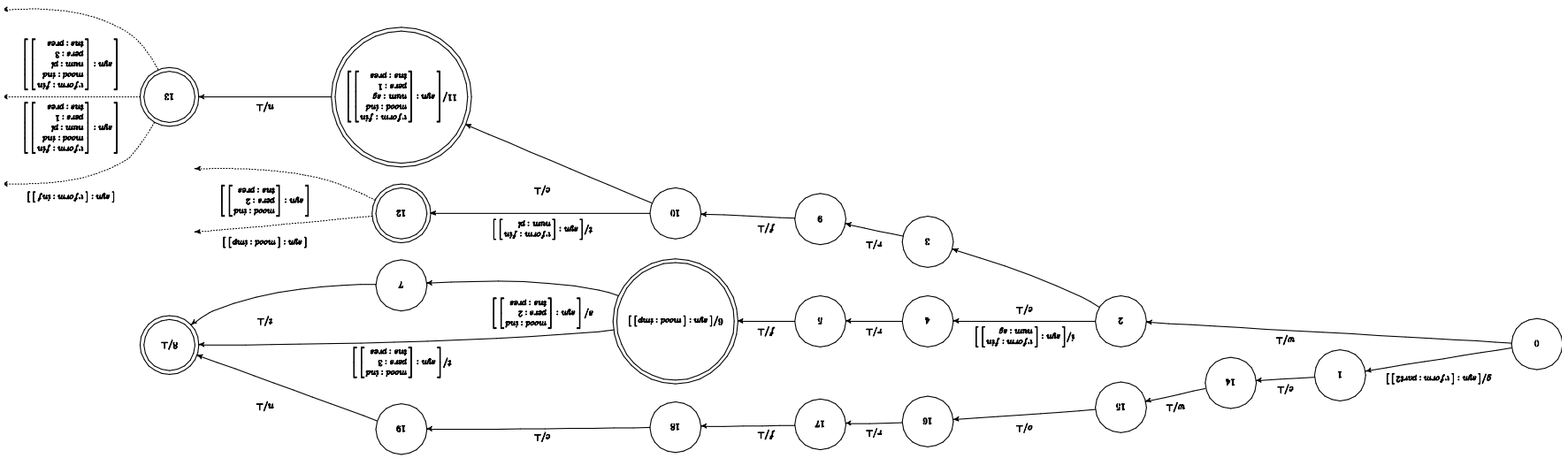
Compiling lexicons

```
FSM<2.0> interactive interpreter v1.0.0 (tropical semiring)
Type 'help' for help on commands

[0]> load symspec sigma
Symbol specification sigma.sym (119 user symbols, 12 supertypes, 0
categories) loaded in 0.016s
[0]> load lexicon eng_words
Lexicon "eng_words.lex" (79767 lines) loaded in 0.766s
[acceptor, 696281 states, 79765 final states, 696280 transitions]
[1]> determinize
FSM determinized in 0.218s
[acceptor, 199610 states, 79765 final states, 199609 transitions]
[1]> minimize
FSM minimized in 0.171s
[acceptor, 35798 states, 5894 final states, 76206 transitions]
[1]> print words > words.txt
Strings written to file words.txt.
```

Compiling lexicons
(example lexicon over the unification semiring from page 23)

```
FSM<2.0> interactive interpreter v1.0.0 beta (tropical semiring)
Type 'help' for help on commands

[0]> semiring unification
Semiring changed to 'unification'
[0]> load symspec sigma
Symbol specification sigma.sym (40 user symbols, 0 supertypes, 0
categories) loaded in 0s
[0]> load lexicon werfen.lex
Lexicon "werfen.lex" (11 lines) compiled in 0.016s
[weighted acceptor, 57 states, 10 final states, 56 transitions, 0 eps-
transitions]
[1]> optimize
FSM optimized in 0s
[weighted acceptor, 20 states, 5 final states, 21 transitions, 0 eps-
transitions]
[1]> draw werfen ""
Dot file "werfen.dot" created.
```

The resulting WFSA can be seen on the next page.



Compiling lexicons
(example lexicon over the (fsr,msr) key-value semiring from page 18)

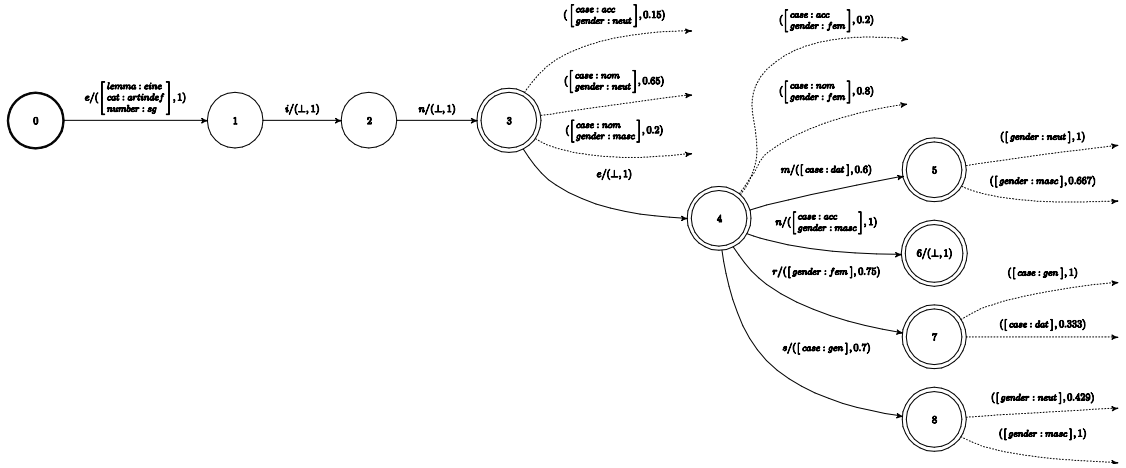
```
FSM<2.0> interactive interpreter v1.0.0 beta (tropical semiring)
Type 'help' for help on commands

[0]> semiring fsr_x_msr
Semiring changed to '(unification,Viterbi)'
[0]> load symspec sigma
Symbol specification sigma.sym (40 user symbols, 0 supertypes,
0 categories) loaded in 0s
[0]> load lexicon ein.lex
Lexicon "ein.lex" (13 lines) compiled in 0.016s
[weighted acceptor, 53 states, 12 final states, 52 transitions,
0 eps-transitions]
[1]> optimize
FSM optimized in 0s
[weighted acceptor, 9 states, 6 final states, 8 transitions,
0 eps-transitions]
```

The lexicon file `ein.lex` lists the indefinite articles in German:

Example: Lexicon with all forms of German indefinite article *ein*
(with some invented probalilities)

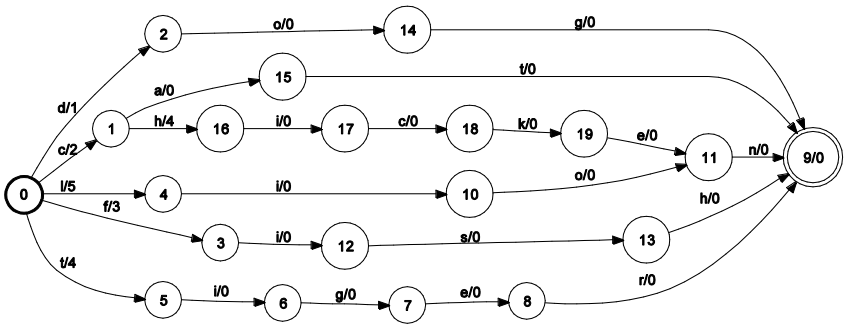
```
eines <([lemma:eine, cat:artindef, number:sg, case:gen, gender:neut],0.30)>
eines <([lemma:eine, cat:artindef, number:sg, case:gen, gender:masc],0.70)>
einer <([lemma:eine, cat:artindef, number:sg, case:dat, gender:fem],0.25)>
einer <([lemma:eine, cat:artindef, number:sg, case:gen, gender:fem],0.75)>
einen <([lemma:eine, cat:artindef, number:sg, case:acc, gender:masc],1)>
einem <([lemma:eine, cat:artindef, number:sg, case:dat, gender:neut],0.6)>
einem <([lemma:eine, cat:artindef, number:sg, case:dat, gender:masc],0.4)>
eine <([lemma:eine, cat:artindef, number:sg, case:acc, gender:fem],0.2)>
eine <([lemma:eine, cat:artindef, number:sg, case:nom, gender:fem],0.8)>
ein <([lemma:eine, cat:artindef, number:sg, case:acc, gender:neut],0.15)>
ein <([lemma:eine, cat:artindef, number:sg, case:nom, gender:neut],0.65)>
ein <([lemma:eine, cat:artindef, number:sg, case:nom, gender:masc],0.2)>
```

The following figure shows the resulting weighted automaton.



Perfect hashing

```
FSM<2.0> interactive interpreter v1.0.0 beta 2 (tropical semiring)
Type 'help' for help on commands

[0]> regex "dog<1>|cat<2>|fish<3>|tiger<4>|lion<5>|chicken<6>"
Regex compiled in 0s
[weighted acceptor, 27 states, 6 final states, 26 transitions]
[1]> optimize
FSM optimized in 0s
[weighted acceptor, 20 states, 1 final states, 24 transitions]
[1]> print words
dog <1>
cat <2>
fish <3>
tiger <4>
lion <5>
chicken <6>
```

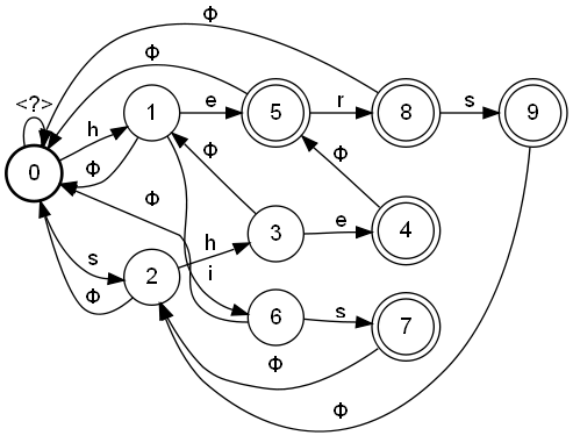


Counting N-grams

```
FSM<2.0> interactive interpreter v0.9.9 (probabilistic semiring)
Type 'help' for help on commands

[0]> load symspec sigma
Symbol specification sigma.sym (118 user symbols, 11 supertypes, 0
categories) loaded in 0s
[0]> load lexicon eng_words
Lexicon "eng_words.lex" (79767 lines) loaded in 0.75s
[acceptor, 696281 states, 79765 final states, 696280 transitions]
[1]> optimize
FSM optimized in 0.5s
[acceptor, 35798 states, 5894 final states, 76206 transitions]
[1]> ngram counter 3
[weighted transducer, 4 states, 1 final states, 590 transitions]
[2]> compose
Intersected/composed FSMs in 0.188s
[weighted transducer, 141532 states, 5840 final states, 332500 transitions]
[1]> project 2
FSM projected in 0s
[weighted acceptor, 141532 states, 5840 final states, 332500 transitions]
[1]> optimize
FSM optimized in 0.75s
[weighted acceptor, 595 states, 3 final states, 7054 transitions]
[1]> lookup "ing" "ate" "tio" "ion"
ing <7259>
ate <2929>
tio <3867>
ion <4527>
[1]>
```

Example: Creating a failure-transition FSA

```
FSM<2.0> interactive interpreter v1.0.0 (tropical semiring)
Type 'help' for help on commands

[0]> load symspec sigma
Symbol specification sigma.sym (26 user symbols, 1 supertypes, 0
categories) loaded in 0.078s

# words.lex contains the entries his, her, hers, she and he
[0]> failure-fsa words.lex
[acceptor, 10 states, 5 final states, 19 transitions]
```



Efficiency tips

Although, most FSM operations are very efficient in theory and practice, dealing with large automata requires a little thought about efficiency:

1. For creating big automata, you'll need a lot of computer memory: 2 GB or better 4 GB.
2. The operations of composition and intersection become more efficient if the two operands are label sorted in the right way: the first operand should be sorted by output symbol, the second one by input label (see `sort`)
3. Optimize early: optimize intermediate FSMs from time to time, especially before during complicated intersections or compositions.
4. Use table format whenever possible (see `convert`). When sorted in an appropriate way (see 1.) composition and intersection work much faster on FSMs in table format.
5. For symbol specifications containing several ten or even hundred thousands of symbols use the precompiled format (created with the `print symspec` command and put to use with `load symspec FILE precompiled`).
6. Use `compose3` instead of `compose` for (edit-)distance computations or other compositions which involve three FSMs.
7. (Insertion) rules and right contexts: When using insertion rules of the form

   ```
   [] -> β / _ RC
   ```

   avoid complicated expressions in the right context: β will be inserted at every position and the correct right context will be checked afterwards which may lead to hopeless inefficiency. Consider matching in reverse in these cases, thereby converting right contexts to much more efficient left contexts.

Limitation and known bugs

1. The XML-parser (used for loading FSM specifications in XML format) should have better mechanisms to point out ill-formed input.
2. The precedence order of the operators is somewhat non-standard.
3. The console program could have more options controlling its behaviour.
4. The commands `compose` and `intersect` may loop in the presence of <phi>-loops.

Exit codes

Upon termination, *fsm2* uses the following exit codes to transmit information to the operation system:

Code	Meaning
`0`	Normal termination
`1`	User break
`2`	Some error occurred
`3`	Invalid script file specified at command line